The background of the slide is a light gray gradient, decorated with numerous realistic water droplets of various sizes. Some droplets are clustered at the top left, while others are scattered across the bottom right and center. The droplets have highlights and shadows, giving them a three-dimensional appearance.

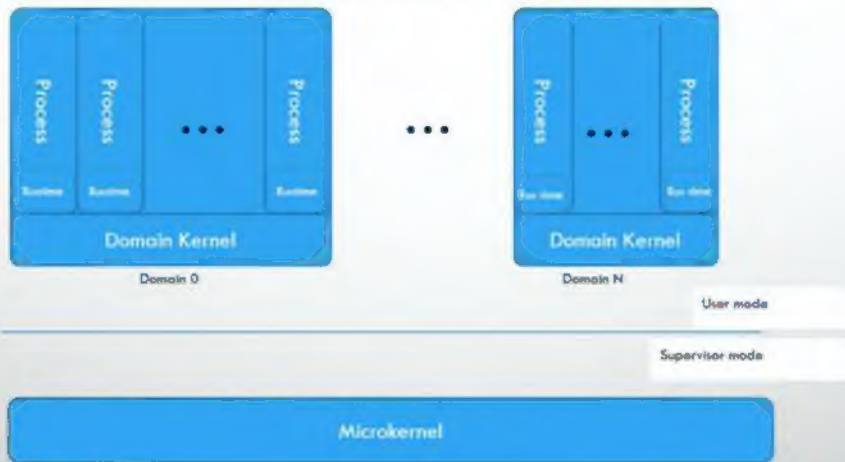
Security Principles and Practices in Midori

Jinsong Yu
January 2014

BRIEF HISTORY OF MIDORI

- System Incubation Effort
- First sprint dated 2006
- Singularity heritage
- Joined OSG in late 2013

SYSTEM OVERVIEW



1. PERVASIVE MANAGED CODE

- Vast majority of the system is written in safe M# code
 - M# language is C# augmented to add immutability and concurrency safety guarantees
- C/C++, ASM, unsafe M# used in a few places:
 - Bottom layers: entire microkernel, part of domain kernel, GC and part of runtime.
 - Unmanaged code we picked up and plan to use in long term: Enigma, ACPI.
 - Unmanaged code we picked up as scaffolding: some multimedia components.

IMPLICATIONS

- Memory safety and type safety
- No buffer overrun, and heap/stack corruption
- No double-free
- No use-after-free
- Allows software-isolated-processes

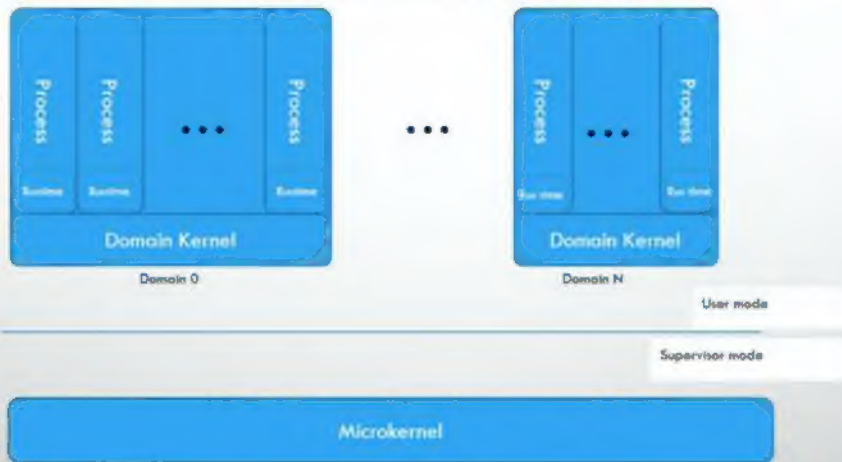
2. CLOSED PROCESSES

- All code in a process is known at compile time and loaded before the process starts.
- No dynamic DLL loading (LoadLibrary)

Implications

- Allows comprehensive compile-time and installation-time checking
- Enables capability based security model (more details later)
- Removes certain run-time failure modes

SYSTEM OVERVIEW



2. CLOSED PROCESSES

- All code in a process is known at compile time and loaded before the process starts.
- No dynamic DLL loading (LoadLibrary)

Implications

- Allows comprehensive compile-time and installation-time checking
- Enables capability based security model (more details later)
- Removes certain run-time failure modes

3. STRICT ERROR MODEL – CHECKED ARITHMETIC

- Checked arithmetic by default, overflow causes immediate process termination

```
bool IsValidRange(uint start, uint count, uint bound)
{
    return start + count <= bound; // terminates if start+count overflows
}
```

- Use unchecked keyword to override

```
bool IsValidRange(uint start, uint count, uint bound)
{
    uint end = unchecked(start + count);
    return end >= start && end <= bound;
}
```

- Same applies to number conversions and divide by 0.
- Helper functions TryAbs(), TryDiv(), etc.

3. STRICT ERROR MODEL - PRECONDITIONS

- Precondition errors cause immediate process termination

```
public void SetCapacity(int count)
    requires count > 0
{ ... }
```

- Preconditions are part of API signature
- Callers are responsible to sanitize parameter values
- TryFoo(...) pattern for parsers and decoders

3. STRICT ERROR MODEL

– NO OUT-OF-BAND EXCEPTIONS

- No dynamic DLL loading, therefore no `BadImageFormatException`
- No threads therefore no `ThreadAbortException`
- Out-of-memory causes immediate process termination
- Stack overflow is same as out-of-memory
- Array bound check error causes immediate process termination
- Dereferencing NULL causes immediate process termination
- Access violation (non-NULL) causes immediate domain termination
- All exceptions are thrown by "throw" statements in the code

3. STRICT ERROR MODEL - DECLARED EXCEPTION

```
public throws int Foo(int count) { ... }
```

```
void Bar() {  
    int a = Foo();      // compile error, exceptions not handled  
    int b = try Foo();  // compile error, attempt to propagate  
                        //      exception while Bar() is not "throws"  
    try { int c = Foo(); } catch { ... }    // OK  
    Result<int> d = trycatch Foo();         // OK  
}
```

```
throws void Bar() {  
    int b = try Foo();  // OK, propagates exception up  
}
```

IMPLICATIONS

- Defaults to strict behavior
 - Prefers deterministic process termination over continue running with inconsistent state
 - Easier to DoS, harder to go beyond DoS
 - Easier to diagnose
-
- All exceptions are explicitly declared and thrown
 - Callers of "throws" code must explicitly handle or propagate exceptions
 - Hard to ignore exceptions
 - Preconditions and throws are part of API signature and therefore versioning story

4. NO SHARED MUTABLE MEMORY CROSS PROCESS BOUNDARY

- Memory shared among different processes must be immutable, guaranteed by type system and compiler.
- Mutable memory is always exclusively owned

Implications

- No TOCTOU issues cross process boundary
- Hard for a process to exploit bugs in another process

5. CONCURRENCY SAFETY – NO THREADS

- Public APIs don't contain any of these:
 - `CreateThread()`
 - `Lock`, `Semaphore`, `AutoResetEvent`, `Monitor`, ...
 - `InterlockedCompareExchange`
- Achieve concurrency by:
 - Decompose or scale out to multiple processes
 - Use safe data-parallelism and task-parallelism APIs
- No instruction-level thread interleaving concerns
- Interleaving only happens at “turn” boundary, which is clearly visible in source code

5. CONCURRENCY SAFETY - EXAMPLE

```
throws Awaitable<void> FinishPOST(Request request,
    Response response, String filePath, Uint myCookie)
{
    String cookieStr = "Invariant FormatUInt32(myCookie),
    InputStream stream,
    try {
        stream = (try Awaitable GetFileContents(_rootStatic, filePath)) Extract(),
    }
    catch {
        try Awaitable WriteErrorResponse(request, response, HttpStatus.NotFound,
            "Couldn't open file " + filePath),
        return,
    }

    response.SetCookie("my_cookie" + cookieStr),
    try Awaitable WriteFileResponse(request, response, in stream,
```


5. CONCURRENCY SAFETY – TSE

- Language features to guarantee deep immutability of an object graph:
 - immutable / readable / writable / isolated
- Only immutable objects can be shared across data-parallelism and task-parallelism boundaries
 - Corollary all static fields are immutable
- Allows fine-grained reasoning of state invariants and state changes in the code

IMPLICATIONS

- No instruction level thread interleaving concerns
- No TOCTOU and races cross data-parallelism and task-parallelism boundary
- State interleaving only happens at turn boundary in a single-threaded fashion
- TOCTOU inside the same execution unit is easy to reason about
- Language and compiler helps reasoning about state changes

6. CAPABILITY BASED SECURITY MODEL

- Capability vs. ACL

- ACL example:

```
void Foo()  
{  
    File f = File.Open("c:\\windows\\system32\\a.txt"); // May fail due to access denied  
    ...  
}
```

- Capability example:

```
void Foo(File f)  
{  
    ...  
}
```

6 CAPABILITY BASED SECURITY MODEL

- Consuming CPU and Memory resources does not require a capability
- All other resource consumption requires capability. No static methods to access resources
- None of these exists:
 - `File.Open()`, `File.Create()`, ...
 - `DateTime.Now()`, `CTime::GetCurrentTime()`, ...
 - `Process.GetProcessByName()`
- Capability provisioning rolls all the way up to process entry point
 - For program entry point, capabilities are supplied by the system, controlled at installation time
 - For child processes, capabilities are supplied by the parent process

6 CAPABILITY BASED SECURITY MODEL - EXAMPLE

```
sealed class PerfInfo : AShellAddin
{
    [Remotable(IsProgram = true)]
    public PerfInfo(
        PerfCounterProviderRepository directory,
        SystemPerfCounterQuery systemPerfCounterQuery,
        ProcessPerfCounterQuery processPerfCounterQuery,
        ProcessInformation processInformation,
        AsyncFactory parentAsyncFactory,
        Clock clock)
    {
        ...
    }
    ...
}
```